Resolution of Constraints in Algebras of Rational Trees

Luis Damas, Nelma Moreira, Sabine Broda
{luis,nam,sbb}@ncc.up.pt

LIACC,Universidade do Porto
Rua do Campo Alegre, 823, 4100 Porto
Portugal

Abstract. This work presents a constraint solver for the domain of rational trees. Since the problem is NP-hard the strategy used by the solver is to reduce as much as possible, in polynomial time, the size of the constraints using a rewriting system before applying a complete algorithm. This rewriting system works essentially by rewriting constraints using the information in a partial model. An efficient C implementation of the rewriting system is described and an algorithm for factoring complex constraints is also presented.

1 Introduction

The topic of constraints over syntactic domains has raised a considerable interest in the Computational Linguistics community. As a matter of fact constraints over the algebra of rational trees can significantly contribute to the reduction of the search space of problems in NLP while increasing the expressive power of formalisms for NLP [DMV91,DV89].

Unification-based grammar formalisms ([SUP$^+$83], [Usz86],[KB82], [PS87], etc.) describe linguistic information by means of constraints over *feature structures*, which are basically sets of attribute-value pairs, where values can be atomic symbols or embedded feature structures, e.g.

$$[cat = np, agr = [num = sg, pers = 3rd]]$$

These structures and their combination can be seen as conjunctions of equality constraints which satisfiability can be tested by efficient unification algorithms. But the extension of these formalisms to express complex constraints involving negation and disjunction of equality constraints, besides arising formal theoretical problems leads to a NP-hard satisfiability problem.

From a formal point of view the problem is well understood after the foundational works of [RK86,Smo89] establishing Feature Logics. It turns out that the standard model for feature logics, namely rational trees, has a close relationship with the standard algebra of rational trees in Logic Programming and with its complete axiomatization presented in [Mah88]. As a matter of fact it can be proved [DMV92] that the satisfiability problem for the complete axiomatization

of feature logics can be reduced to the satisfiability problem for Maher complete axiomatization of the algebra of rational trees.

From a practical point of view the fact that the satisfiability problem is NP-hard tends to manifest itself in a dramatic way in practical applications motivated several specialized algorithms to minimize this problem [Kas87,ED88,MK91].

In [DV92] it was argued that any practical approach to the satisfiability problem should use factorization techniques to reduce the size of the input formulae to which any complete algorithm for satisfiability is applied, since such factorization can reduce by an exponential factor the overall computational cost of the process. In that work a rewrite system, working in polynomial time, was used to factor out deterministic information contained in a complex constraint and simplify the remaining formula using that deterministic information. In [DMV92] that rewrite system was extended to a complete rewrite system for satisfiability which avoided as much as possible the multiplication of disjunctions which is the origin of the NP-hardness of the satisfiability problem.

In this work we present a solver for constraints over the domain of rational trees using the rewrite system mentioned above.

The rest of this paper proceeds as follows. We start by defining our constraint language, which was designed to enable the introduction of equality constraints on terms or rational trees without using any quantifiers, in section 2. In section 3 we define a complete rewrite system for expressions in the constraint language. In section 4 we present in some detail the low-level implementation of part of the rewrite system. In section 5 we will present an algorithm for factoring out of two constraints any deterministic information which is common to both.

2 The Constraint Language

Consider the first order language $\mathcal{L}$ built from the countable sets of variables $Vars = \{x, y, z, \ldots\}$, function symbols $F = \{f, g, h, \ldots\}$ and equality as the only predicate symbol. As usual, the functional symbols of arity 0 will be denoted by a, b, $\ldots$ and will be referred to as atoms. For this language Maher presented a complete axiomatization of the algebra of rational trees, $\mathcal{RT}$, [Mah88]. This theory of $\mathcal{RT}$ is complete in the sense that a sentence is valid in $\mathcal{RT}$ if and only if it is a logical consequence of the theory. We now introduce a quantifier free constraint language, in which the formulae of $\mathcal{L}$ will be encoded. Here function symbols appear, whenever necessary, with their arities and are then denoted by f^n, while f^n_i stands for the ith projection of a function symbol f of arity n. The letters s and t will always denote variables or atoms. Expressions of the form $x.f^n_i$ will be called *slots*. We define the *constraints* of the language by:

$$c ::= t.f^n \mid t = t \mid t.f^n_i \doteq t,\ 1 \leq i \leq n \mid$$
$$false \mid true \mid \neg c \mid c \wedge c \mid c \vee c$$

Note that one can look at each constraint c as an abbreviation for a formula of $\mathcal{L}$, interpreting $t.f^n$ as $\exists z_1 \ldots \exists z_n\ t = f(z_1, \ldots, z_n)$ and $t.f^n_i \doteq s$ as $\exists z_1 \ldots \exists z_n\ t =$

$f(z_1, \ldots, z_n) \wedge z_i = s$. On the other hand there exists an equivalent constraint c for every formula of $\mathcal{L}$. To see this, recall that Maher proves in [Mah88] that any of these formulae is equivalent to a boolean combination of *rational basic* formulae, for which it is easy to find an equivalent constraint. Finally notice the similarity of this constraint language and the Smolka's Feature Logics [Smo89] with f_i^n playing the role of features. More recently Smolka and others [ST92] introduced a Feature Tree Logic which includes sort and arity constraints much similar to our $x.f^n$ constraint.

We call the first five types of constraints defined above atomic constraints and say that a set of atomic constraints $\mathcal{M}$, denoting their conjunction, is in *solved form* if and only if it satisfies the following conditions:

1. every constraint in $\mathcal{M}$ is of one of the forms $x.f^n$, $x.f_i^n \doteq t$ or $x = t$;
2. if $x = t$ is in $\mathcal{M}$, then x occurs exactly once in $\mathcal{M}$;
3. if $x.f_i^n \doteq t$ and $x.f_i^n \doteq s$ are in $\mathcal{M}$, then t is equal to s;
4. if $x.f_i^n \doteq t$ is in $\mathcal{M}$, then $x.f^n$ is also in $\mathcal{M}$;
5. if $x.f^n$ is in $\mathcal{M}$, then there is no constraint in $\mathcal{M}$ of the form $x.g^m$;
6. if for some x, y and t, both $x.f_i^n \doteq t$ and $y.f_i^n \doteq t$ are in $\mathcal{M}$, then for some j between 1 and n, there is no s, such that $x.f_j^n \doteq s$ and $y.f_j^n \doteq s$ are both in $\mathcal{M}$.

The purpose of the last clause in the previous definition is to force solved forms to contain $x = y$, whenever for some f^n and every i between 1 and n, there is s_i, such that $x.f_i^n \doteq s_i$ and $y.f_i^n \doteq s_i$ hold.

It is easy to prove that solved forms are satisfiable and that every set of atomic constraints $\mathcal{M}$ can be reduced in quadratic time to an equivalent set, which is either in solved form or equal to $\perp$, using the following set of simplification rules, that correspond to the Herbrand rules for solving equations in a first order logic.

1. $\{true\} \cup \mathcal{M} \rightarrow \mathcal{M}$
2. $\{false\} \cup \mathcal{M} \rightarrow \perp$
3. $\{t = t\} \cup \mathcal{M} \rightarrow \mathcal{M}$
4. $\{a = b\} \cup \mathcal{M} \rightarrow \perp$
5. $\{x = t\} \cup \mathcal{M} \rightarrow \{x = t\} \cup [t/x]\mathcal{M}$ if x is not equal to t and x occurs in $\mathcal{M}$
6. $\{a = x\} \cup \mathcal{M} \rightarrow \{x = a\} \cup \mathcal{M}$
7. $\{x.f_i^n \doteq t, x.f_i^n \doteq s\} \cup \mathcal{M} \rightarrow \{x.f_i^n \doteq t, t \doteq s\} \cup \mathcal{M}$
8. $\{a.f_i^n \doteq t\} \cup \mathcal{M} \rightarrow \perp$
9. $\{a.f^n\} \cup \mathcal{M} \rightarrow \perp$
10. $\{x.f_i^n \doteq t\} \cup \mathcal{M} \rightarrow \{x.f^n, x.f_i^n \doteq t\} \cup \mathcal{M}$ if $x.f^n \notin \mathcal{M}$
11. $\{x.f^n, x.g^m\} \cup \mathcal{M} \rightarrow \perp$
12. if $x = y \notin \mathcal{M}$, but for all $1 \leq i \leq n$ there exists t_i such that $x.f_i^n \doteq t_i \in \mathcal{M}$ and $y.f_i^n \doteq t_i \in \mathcal{M}$, then $\mathcal{M} \rightarrow \{x = y\} \cup \mathcal{M}$.

3 Rewriting System

From now on let $\mathcal{M}$ be a solved form and $\mathcal{C}$ a finite set of constraints representing their conjunction. We say that $\mathcal{M}$ is a *partial model* of $\mathcal{C}$ if and only if every model of $\mathcal{C}$ is a model of $\mathcal{M}$. When every model of $\mathcal{M}$ is a model of $\mathcal{C}$, but no proper subset of $\mathcal{M}$ satisfies this condition, we will say that $\mathcal{M}$ is a *minimal* model of $\mathcal{C}$. By using disjunctive forms it can be proved that any set of constraints $\mathcal{C}$ admits at most a finite number of minimal models.

Our rewriting system produces from a set of constraints $\mathcal{C}_0$ a partial model $\mathcal{M}$ and a smaller set of constraints $\mathcal{C}$, such that any minimal model of $\mathcal{C}_0$ can be obtained by conjoining (i.e. "unifying") a minimal model of $\mathcal{C}$ with $\mathcal{M}$ and moreover for any minimal model of $\mathcal{C}$ the union $\mathcal{M} \cup \mathcal{C}$ is satisfiable. The rewriting system for pairs $\langle \mathcal{M}, \mathcal{C} \rangle$ is defined by the following rules:

$$\langle \mathcal{M}, \mathcal{C} \cup \{false\} \rangle \rightarrow \langle \bot, \emptyset \rangle$$
$$\langle \mathcal{M}, \mathcal{C} \cup \{true\} \rangle \rightarrow \langle \mathcal{M}, \mathcal{C} \rangle$$
$$\langle \mathcal{M}, \mathcal{C} \cup \{x = t\} \rangle \rightarrow \langle \mathcal{M} \cup \{x = t\}, \mathcal{C} \rangle$$
$$\langle \mathcal{M}, \mathcal{C} \cup \{x.f^n\} \rangle \rightarrow \langle \mathcal{M} \cup \{x.f^n\}, \mathcal{C} \rangle$$
$$\langle \mathcal{M}, \mathcal{C} \cup \{x.f_i^n \doteq t\} \rangle \rightarrow \langle \mathcal{M} \cup \{x.f_i^n \doteq t\}, \mathcal{C} \rangle$$

with the convention that after each application of one of the rewrite rules the new partial model is reduced to solved form and the resulting set of constraints is closed under $\longrightarrow_{\mathcal{M}}$ as defined below.

The complete set of rewrite rules $\longrightarrow_{\mathcal{M}}$ for terms and constraints follows:

$x \longrightarrow_{\mathcal{M}} t$	if $x = t \in \mathcal{M}$
$x.f_i^n \longrightarrow_{\mathcal{M}} t$	if $x.f_i^n \doteq t \in \mathcal{M}$
$c \longrightarrow_{\bot} false$	
$\neg true \longrightarrow_{\mathcal{M}} false$	
$\neg false \longrightarrow_{\mathcal{M}} true$	
$\neg\neg c \longrightarrow_{\mathcal{M}} c$	
$\neg(c_1 \wedge c_2) \longrightarrow_{\mathcal{M}} \neg c_1 \vee \neg c_2$	
$\neg(c_1 \vee c_2) \longrightarrow_{\mathcal{M}} \neg c_1 \wedge \neg c_2$	
$true \wedge c \longrightarrow_{\mathcal{M}} c$	
$false \wedge c \longrightarrow_{\mathcal{M}} false$	
$c \wedge true \longrightarrow_{\mathcal{M}} c$	
$c \wedge false \longrightarrow_{\mathcal{M}} false$	
$true \vee c \longrightarrow_{\mathcal{M}} true$	
$false \vee c \longrightarrow_{\mathcal{M}} c$	
$c \vee true \longrightarrow_{\mathcal{M}} true$	
$c \vee false \longrightarrow_{\mathcal{M}} c$	
$(c_1 \wedge c_2) \wedge c_3 \longrightarrow_{\mathcal{M}} c_1 \wedge (c_2 \wedge c_3)$	
$a = b \longrightarrow_{\mathcal{M}} false$	if a and b are distinct atoms
$a = x \longrightarrow_{\mathcal{M}} x = a$	
$t = t \longrightarrow_{\mathcal{M}} true$	
$x.f^n \longrightarrow_{\mathcal{M}} true$	if $x.f^n \in \mathcal{M}$

$x.f^n \ \longrightarrow_{\mathcal{M}} \ false$	if $x.g^m \in \mathcal{M}$
$a.f^n \ \longrightarrow_{\mathcal{M}} \ false$	
$a.f_i^n \doteq t \ \longrightarrow_{\mathcal{M}} \ false$	
$x = t \ \longrightarrow_{\mathcal{M}} \ false$	if $\mathcal{M} \cup \{x = t\} \rightarrow \bot$
$x = t \wedge c \ \longrightarrow_{\mathcal{M}} \ x = t \wedge c'$	if $c \ \longrightarrow^{\star}_{\mathcal{M} \cup \{x=t\}} \ c'$
$x.f_i^n \doteq t \ \longrightarrow_{\mathcal{M}} \ false$	if $\mathcal{M} \cup \{x.f_i^n \doteq t\} \rightarrow \bot$
$x.f_i^n \doteq t \wedge c \ \longrightarrow_{\mathcal{M}} \ x.f_i^n \doteq t \wedge c'$	if $c \ \longrightarrow^{\star}_{\mathcal{M} \cup \{x.f_i^n \doteq t\}} \ c'$
$x.f^n \wedge c \ \longrightarrow_{\mathcal{M}} \ x.f^n \wedge c'$	if $c \ \longrightarrow^{\star}_{\mathcal{M} \cup \{x.f^n\}} \ c'$
$x \neq t \wedge c \ \longrightarrow_{\mathcal{M}} \ c \wedge x \neq t$	if any non-negated equality occurs in c
$x.f_i^n \not\doteq t \wedge c \ \longrightarrow_{\mathcal{M}} \ c \wedge x.f_i^n \not\doteq t$	if any non-negated equality occurs in c
$\neg x.f^n \wedge c \ \longrightarrow_{\mathcal{M}} \ c \wedge \neg x.f^n$	if any non-negated equality occurs in c
$(c_1 \vee c_2) \wedge c_3 \ \longrightarrow_{\mathcal{M}} \ (c_1 \wedge c_3) \vee (c_2 \wedge c_3)$	if both c_1 and c_2 are $\mathcal{M}$-dependent with c_3.

Note that in the rules above $\mathcal{M} \cup \mathcal{C}$ denotes the solved form of the union of $\mathcal{M}$ and $\mathcal{C}$, if one exists, or $\bot$ if that union is not satisfiable. The last rule must apply only when both c_1 and c_2 have variables in common with c_3, eventually through "bindings" in $\mathcal{M}$. In order to formalize this notion we need the following definition. Given two constraints c_1 and c_2 and a model $\mathcal{M}$, c_1 and c_2 are $\mathcal{M}$-*dependent* if and only if $Var_{\mathcal{M}}(c_1) \cap Var_{\mathcal{M}}(c_2) \neq \emptyset$, where $Var_{\mathcal{M}}(c)$ is the smallest set satisfying:

if $x \in c$, then $x \in Var_{\mathcal{M}}(c)$;
if $x \in Var_{\mathcal{M}}(c)$ and $x.f_i^n \doteq z \in \mathcal{M}$, then $z \in Var_{\mathcal{M}}(c)$.

Given an initial set of constraints $\mathcal{C}_0$ we apply the rewriting system to $\langle \emptyset, \mathcal{C}_0 \rangle$ to obtain $\langle \mathcal{M}, \mathcal{C} \rangle$. It is easy to prove that $\mathcal{C}_0$ (more precisely the conjunct of all the constraints in $\mathcal{C}_0$) is equivalent to $\mathcal{M} \cup \mathcal{C}$. As a matter of fact this follows from the fact that each rewrite rule is associated with a similar meta-theorem of First Order Logic and/or the axioms of $\mathcal{RT}$.

A proof that all the minimal models of $\mathcal{C}_0$ are obtained by conjoining $\mathcal{M}$ with those of $\mathcal{C}$ follows along similar lines as the proof in [DV92] for feature logics.

The other interesting property of the rewriting system above is that it is complete in the sense that $\langle \mathcal{M}, \mathcal{C} \rangle$ is satisfiable, unless it produces $\bot$ as the final model. The simple (but tedious) proof of this result uses induction. Completeness is achieved mainly by the last rule above for $\longrightarrow_{\mathcal{M}}$. However, even if this rule attempts to limit the number of cases where it applies to an essential minimum, it causes NP-completeness of the rewriting process since it can lead to an exponential growth of the constraints. If we omit this rule, then the rewrite process becomes polynomial, although incomplete. As a technique to decrease the number of times the rule is used, one can treat disjunctions $C_0^1 \vee C_0^2$ in the following way: First apply the rewrite system to each C_0^i obtaining partial models $\mathcal{M}_i$ and smaller sets of constraints C^i. Then push redundancies out of $\mathcal{M}_1$ and $\mathcal{M}_2$ using the algorithm described in the section 5 and obtain sets COM, $\tilde{\mathcal{M}}_1$ and $\tilde{\mathcal{M}}_1$, such that each $\mathcal{M}_i$ is equivalent to $COM \wedge \tilde{\mathcal{M}}_i$. Finally substitute $C_0^1 \vee C_0^2$ by $COM \wedge (\tilde{\mathcal{M}}_1 \cup C^1 \ \vee \ \tilde{\mathcal{M}}_2 \cup C^2)$.

4 Implementation

In this section we present a constraint solver based on the rewriting system described above. Note that given a solved form $\mathcal{M}$, and considering that the equality is a equivalence relation, $\mathcal{M}$ can be partitioned into equivalence classes. Given an order $<_{\mathcal{T}}$ on terms[1] we can induce an order in these classes and so, to each set of satisfiable atomic constraints corresponds a set of normalized classes. A set $\mathcal{N} = \{l_1, \ldots, l_n\}$, is a normalized solved form iff:

1. each l_i is of the form $v_1 = \ldots = v_k$, where each v_i is a variable, an atom or a slot and $v_i <_{\mathcal{T}} v_j$, for $i < j \in \{1, \ldots, k\}$, or l_i is $x.f_i^n$.
2. each l_i has at most an atom and in that case it is the first element.
3. if $v \in l_i$ then v occurs exactly once in $\mathcal{M}$.
4. if l_i is $x.f^n$, x does not occur in other l_j of this form.
5. if x is in l_i and the first element of l_i is an atom then there is no l_j of the form $x.f^n$, for any f and n.
6. if $x.f_i^n \in l_k$ then there is a l_j of the form $x.f^n$.

With a slight modification of rule 5 in section 2 to deal with ordered variables, it is easy to see that each satisfiable constraint can be reduced to an unique equivalent normalized solved form. We will describe an algorithm that given a set of atomic constraints returns a solved form as a set of equality classes or $false$ if the set is not satisfiable. The main features of the algorithm are implemented in `C` but an interface to Prolog is provided via a set of basic predicates. The complete rewrite system was written in Prolog using these predicates. The `C` component of the solver implements essentially the unification of solved forms in a way which is very similar to the Prolog implementation of unification. The main reason for a detailed presentation is the novel use we made of the trail mechanism which is not only used to recover a previous state but also to produce, as a solved form, the "differences" between the current state and the previous state (see subsection 4.3).

4.1 Rational Tree Representation

The representation of rational trees to be used, allows not only an efficient implementation of unification but also provides an incremental way of obtaining partial models, which is suitable for the contexted rewrite of inner disjunctions of a complex constraint. Given a set of atomic constraints, a destructive unification algorithm is used, while producing a trail, and then undoing unification, by also using the trail, we retrieve the associated solved form (partial model).

Besides variables and atoms, the notion of term is extended to objects of the form f^n, denoting a functor (function symbol) f with arity n, and to *slots* of the

[1] Let $Vars$ and F be provided with the lexicographical order and let variables, atoms and slots be terms. Then consider the following order $<_{\mathcal{T}}$ on terms: atoms are less than variables, and variables less than slots; two atoms or two variables are compared lexicographically; two slots are first compared by their variables, if equal then by their functor and arity and finally if every thing else is identical by their projections.

form $x.f_i^n$ [2]. Terms will be stored in a table where each one is a structure with the following fields:

`kind` which can have the values $AtomS$, $VarS$, $FuncS$ or $SlotS$ indicating that the term is an *atom*, a *variable*, a *functor* or a *slot* respectively. These values are sorted by increasing order.

`name` if the term corresponds to an atom or a variable this field is their identifier (a Prolog atom in the actual implementation); if the term is a slot, it is the projection identifier.

`value` a link to the terms in the same equality class or NIL.

`daughters` if the term is a variable or a slot this field is a link to its subtrees; otherwise its value is NIL.

`base` if the term is a slot $x.f_i^n$ this is a pointer to the entry corresponding to the variable x; if the term is a variable, it can be a pointer to a functor term.

`next` link to the next term in the table.

A partial model is represented on the `trail`. The `trail` is a stack that contains pointers to the terms which `value` have changed during the rewrite process. Two pointers `TrailOld` and `TrailPtr` will mark the beginning and the ending of the portion of stack currently in use.

4.2 Solved Form Algorithm

The basic algorithm for the unification of two terms is given in figure 2. As usual substitutions are replaced by a bind/dereference mechanism, so before any two terms are unified they must be dereferenced, see figure 1. In the unification procedure whenever the `value` of a term is bounded, its pointer is added to the top of the `trail`, see figure 1, and in this way the active model is extended. Whenever two variables $v1$ and $v2$ (or a variable and a slot) are unified we must ensure that all subtrees of $v2$ share with subtrees of $v1$. This is done by the procedure `UnifySubTrees`, figure 3. As new terms maybe added to the trail, this can lead to some redundancies which will be eliminated when the solved form will be retrieved. The rest of the algorithm is basically the implementation of the simplification rules given in section 2.

The following Prolog predicates are provided to rewrite atomic constraints (the number after the slash indicates its arity).

`add_ac_va/2` add a constraint of the form $x = a$
`add_ac_vv/2` add a constraint of the form $x = x$
`add_ac_fa/5` add a constraint of the form $x.f_i^n \doteq a$
`add_ac_fv/5` add a constraint of the form $x.f_i^n \doteq x$
`add_ac_f/3` add a constraint of the form $x.f^n$

[2] Recall that it represents the ith projection of x which main functor is f of arity n

```
Deref(Term v)
{   while(v->value) v=v->value;
    return v;
}
Bind(Term v1,Term v2)
{   v1->value=v2;
    *TrailPtr++=v1;
}
```

Fig. 1. *Dereference of a term and bind of two terms.*

```
Unify(Term v1,Term v2)
{   if(v1->kind>v2->kind) { /* exchange v1 with v2 */
        Term t=v1; v1=v2; v2=t; }
    if(v1->kind==AtomS)
        if(v2->kind==AtomS) return v1==v2;
        if(v2->kind==VarS) {
            /* test if v2 is not bounded to a functor */
            if(IsFunctor(v2)) return 0;
                Bind(v2,v1);
                return 1;
            }
        if(v2->kind==SlotS) {
            /* test if v2 is not of the form a.f_i^n */
            if(!IsProper(v2)) return 0;
            Bind(v2,v1);
            return 1;
         }
    if(v1->kind==VarS) {
        /* v2 is a variable or slot */
        if(v1==v2) return 1;
        if(!SameFunctor(v1,v2))return 0;
        if(!IsProper(v2)) return 0;
        /* sort variables */
        if(Compare(v2,v1)) { exchange v1 with v2 */
            Term t=v1; v1=v2; v2=t; }
        Bind(v2,v1);
        return UnifySubTrees(v1,v2);
    }
}
```

Fig. 2. *Unification algorithm.*

```
UnifySubTrees(Term u, Term v)
{   Term du ,dv;
    if(!SameFunctor(v,u))return 0;
    /* ensure all daughters of v share with daughters of u */
    du =u;
    dv = v->daughters;
    while(dv!=NIL) {
        while(du->daughters!=NIL
                && du->daughters->name < dv->name)
            du = du->daughters;
        /* if u does not have that subtree it will be create */
        if(du->daughters==NIL ||du->daughters->name!=dv->name) {
            Term t = (Term) tmp_alloc(sizeof(*t));
            t->kind = SlotS;
            t->name=dv->name;
            t->base = u;
            t->value = NIL;
            t->daughters= du->daughters;
            du->daughters= t;
        }
        /* unify correspondent daughters of u and v */
        if(!Unify(Deref(du->daughters),Deref(dv)) return 0;
        dv=dv->daughters;
    }
    return 1;
    }
}
```

Fig. 3. *Unification of subtrees.*

For each argument the associated term is looked up in the `term table` and if not found, is created and added to the table [3]. Then the `Unify` procedure is called and if it fails, the predicate will fail. The last predicate `add_ac_f` is a bit different because instead of binding the `value` of x with the term representing f^n we just bind it to the `base` of x^4. The reason is that two different terms can have the same functor. In this case they will be equal only if all their subtrees are defined and equal.

When no more constraints are to be added (and no failure has occurred) a call to the predicate `undo_ac/1`, see figure 4, returns a solved form as a set of equality classes. According to the rule 12 in section 2, every two terms that agree in all their subtrees (slots) are unified. Then, beginning at the top of the `trail` each term is dereferenced and all the terms that dereference to the same `value` are removed from the `trail` and, joined in the same class. If a term is a slot, its `base` must be dereferenced[5] and the slot associated to the new `base` is added to the class. That is so, because that term could have been inserted in the trail before its `base` was bounded to another term. This step can also eliminate redundancies created by `UnifySubTrees`, see example below. Finally the values of all terms are zeroed and the `trail` is emptied. To illustrated, let $\mathcal{M}$ be a satisfiable set of atomic constraints: $\{z.f_1^4 \doteq b, z = y, z = x, w = b, x.f_1^4 \doteq u, x.f_1^4 \doteq b\}$. After adding these constraints, the `trail` contains pointers to the terms described in the following table:

`term`	`term.value`
$z.f_1^4$	b
z	y
$y.f_1^4$	b
y	x
$x.f_1^4$	b
w	b
u	b

where the third and fifth elements are due to unification of subtrees and the last one is due to the dereference of $x.f_1^4$. The last conjunct in $\mathcal{M}$ was trivially true, so no more elements were added. This leads to the following normalized solved form: $\{b = u = w = x.f_1^4, x.f^4, x = y = z\}$. Note that if the first two constraints in $\mathcal{M}$ were swapped, then the first element will not appear in the `trail`.

The claim that `undo_ac` returns a normalized solved form of $\mathcal{M}$ follows from the fact that:

– the unification algorithm ensures that $\mathcal{M}$ is satisfiable.

[3] In this stage some clashes can be detected, namely if a variable earlier bounded to an atom is now to be bound to a functor or to a different functor. This avoids some tests done later in the `Unify` procedure.

[4] In the algorithms presented in this paper it is omitted the code concerning the treatment of these constraints.

[5] Function `DerefSlot` accomplishes that.

– every term in the `trail` occurred in an equation of $\mathcal{M}$ or results from the unification of subtrees (akin to application of a substitution); and its value is the other element of the equation or corresponds to one or more applications of rules 5 and 7 of section 2, which preserve equivalence.
– by construction the result of `undo_ac` is a normalized solved form.

```
undo_ac()
{   Term *p,r0,r1;
    SolvForm classes=NIL;
    /* apply rule 12 */
    check_eq_terms();
    while(1){
        *p=TrailPtr;
        r0=*--p;
        /* find first thing left in the trail;
        if a term r has been removed from the
        trail marked(r) will succeed */
        while(p!=TrailOld && marked(r0)) r0=*--p;
        if(p==TrailOld) break; /* nothing left */
        r0=Deref(r0);
        ++p; /* r0 back to the trail */
        StartClass();
        AddToClass(r0);
        /* find another term in the same class */
        while(p!=TrailOld){
            r1=*--p;
            if(marked(r1)) continue;
            if(Deref(r1)!=r0) continue;
            mark(r1); /* remove r1 from the trail */
            /* check slot base */
            r1=DerefSlot(r1);
            AddToClass(r1);
        }
        classes=MkClasses(MkAtomicClass(),classes);
    }
    /* clean trail and term values */
    while(TrailPtr!=TrailOld)(*--TrailPtr)->value=NIL;
    TrailPtr=TrailOld;
}
```

Fig. 4. *Retrieve of a solved form.*

The solved form algorithm for a set of atomic constraints can be summarized as follows:

```
solve(C,M):-
          clean_ac,
          add_constraints(C,C1),
          (C1==false -> M=false; undo_ac(M)).
```

where `C` is a list of atomic constraints, `clean_ac` initializes the `term table` and the `trail`[6] and `add_constraints` for each atomic constraint calls the appropriate predicate and returns $false$ if any of them fails.

4.3 General Algorithm

The above algorithm can be efficiently extended to deal with general constraints and implement the complete rewriting system. The basic idea is to mark the model whenever a disjunction or a negation occurs. In this way all conjunctions of atomic constraints can be treated in a similar manner. Whenever a disjunct is rewritten the solved form corresponding to its atomic part (set of atomic constraints) is extracted, if it is satisfiable. Otherwise the model constructed so far (back to the last mark) must be erased, a new mark must be set and $false$ is produced (for that disjunct or negation). This mechanism is achieved by having a stack of `choice points` which are the trail bounds. To set a `choice point` the current beginning of the trail is added to the top of the `choice point stack` and the current beginning of the trail is reset to be the current ending. The inverse operation is done whenever a model is extracted (`undo_ac` only extracts a model between two `trail` bounds) or a failure occurs, see figure 5. The following Prolog predicates are provided:

`clean_ac/0` set the active model to be the empty model
`mark_ac/0` set a choice point for the model
`undo_ac/1` restore previously marked model and returns a solved form
`fail_ac/0` restore previously marked model

Now the complete rewriting system can be easily implemented in Prolog. Here we just present a small fragment of the program[7].

```
solve(C,C1):-
        clean_ac,
        rewrite(C,C0)
        (C0==C-> C1=C0; solve(C0,C1)).
rewrite(C,C1):-
        rewrite_atomic(C,C0),
        rewrite_m(C0,C1),
        undo_ac(M),and(M,C1,C2).
```

[6] The active model is the empty model.

[7] Negations have been pushed down atomic constraints and the elimination of trivial constraints and earlier detection of failures have been omitted as well as other control features.

```
clean_ac()
{   if(TrailBase==0)
        TrailBase = (Term *) malloc(sizeof(Term)*TRAIL_SIZE);
    TermTable= 0;
    TrailPtr = TrailOld = TrailBase;
    ChoicePtr = ChoicePointBase ;
    return 1;
}

mark_ac()
{   *ChoicePtr++ = TrailOld;
    TrailOld = TrailPtr;
    return 1;
}

fail_ac()
{   while(TrailPtr!=TrailOld) (*--TrailPtr)->value = NIL;
    TrailOld = *--ChoicePtr;
    return 1;
}
```

Fig. 5. *Set and remove choice points.*

```
        ⋮
rewrite_m(and(A,B),C):-
        rewrite_m(A,A1),
        rewrite_m(B,B1).
        and(A1,B1,C).
rewrite_m(or(A,B),C):-
        mark_ac,
        rewrite(A,A1),
        rewrite_tail_or(B,B1),
        or(A1,B1,C).
rewrite_m(not(A),C):-
        mark_ac,
        rewrite(A,A1),
        not(A1,C).
        ⋮
```

The predicate `rewrite_atomic` is similar to `add_constraints` but scans a general constraint and looks for atomic constraints in the "top conjunction". The last rule of the rewrite system $\longrightarrow_{\mathcal{M}}$ in section 3, is applied only when nothing else applies.

5 Common Factor Detection

We now present an algorithm that, given two partial models A and B, constructs models COM, $\tilde{A}$ and $\tilde{B}$ such that $A \vee B$ is equivalent to $COM \wedge (\tilde{A} \vee \tilde{B})$, and such that $\tilde{A}$ and $\tilde{B}$ have no common factors. Remember that every partial model is in particular a solved form. For a solved form $\mathcal{M}$ we define the set of equivalence classes of $\mathcal{M}$ by

$$EQ(\mathcal{M}) = \{[s] : x = s \in \mathcal{M}\},$$

where

$$[s] = \{s\} \cup \{x : x = s \in \mathcal{M}\}.$$

We also define

$$Proj(\mathcal{M}) = \{class(x).f_i^n = class(t) : x.f_i^n \doteq t \in \mathcal{M}\},$$

and

$$Funct(\mathcal{M}) = \{class(x).f^n : x.f^n \in \mathcal{M}\},$$

where

$$class(u) = \begin{cases} [s] & \text{if } u \in [s] \text{ for some } [s] \in EQ(\mathcal{M}) \\ \{u\} & \text{otherwise.} \end{cases}$$

After computing $EQ(A)$, $EQ(B)$, $Proj(A)$, $Proj(B)$, $Funct(A)$ and $Funct(B)$ the algorithm consists of four steps:

(1) First let $COM = \emptyset$ and apply as long as possible the following simplification rule to $EQ(A)$, $EQ(B)$ and COM:

$$\begin{cases} EQ(A) := \{\{s_1, \ldots, u, \ldots, v, \ldots, s_n\}\} \cup Rest(A) \\ EQ(B) := \{\{t_1, \ldots, u, \ldots, v, \ldots, t_m\}\} \cup Rest(B) \\ COM \end{cases}$$

$$\Longrightarrow$$

$$\begin{cases} EQ(A) := \{\{s_1, \ldots, v, \ldots, s_n\}\} \cup Rest(A) \\ EQ(B) := \{\{t_1, \ldots, v, \ldots, t_m\}\} \cup Rest(B) \\ COM := COM \cup \{u = v\} \end{cases}$$

(2) Now apply as long as possible the next rule to $Proj(A)$, $Proj(B)$ and COM:

$$\begin{cases} Proj(A) := \{\{x_1, \ldots, z, \ldots, x_l\}.f_i^n = \{s_1, \ldots, u, \ldots, s_n\}\} \cup Rest\_Proj(A) \\ Proj(B) := \{\{y_1, \ldots, z, \ldots, y_k\}.f_i^n = \{t_1, \ldots, u, \ldots, t_m\}\} \cup Rest\_Proj(B) \\ COM \end{cases}$$

$$\Longrightarrow$$

$$\begin{cases} Proj(A) := Rest\_Proj(A) \\ Proj(B) := Rest\_Proj(B) \\ COM := COM \cup \{z.f_i^n \doteq u\} \end{cases}$$

(3) Apply the next rule to $Funct(A)$, $Funct(B)$ and COM:

$$\begin{cases} Funct(A) := \{\{x_1, \ldots, z, \ldots, x_l\}.f^n\} \cup Rest\_Funct(A) \\ Funct(B) := \{\{y_1, \ldots, z, \ldots, y_k\}.f^n\} \cup Rest\_Funct(B) \\ COM \end{cases}$$

$$\Longrightarrow$$

$$\begin{cases} Funct(A) := Rest\_Funct(A) \\ Funct(B) := Rest\_Funct(B) \\ COM := COM \cup \{z.f^n\} \end{cases}$$

(4) Finally compute $\tilde{A}$ and $\tilde{B}$ by

$$\begin{aligned} \tilde{X} = &\{x.f_i^n \doteq t : class(x).f_i^n = class(t) \in Proj(X)\} \cup \\ &\{x.f^n : class(x).f^n \in Funct(X)\} \cup \\ &\{x = s : x \in [s] \in EQ(X) \text{ and } x \text{ is different from } s\}. \end{aligned}$$

Applying this algorithm to

$$\begin{aligned} A &= \{x_2 = x_1, x_1.f_2^2 \doteq x_4, x_1.f^2, x_5 = x_4, x_6 = x_4, x_4.g^1\} \\ B &= \{x_2 = x_3, x_3.f_2^2 \doteq x_7, x_3.f^2, x_5 = x_7, x_6 = x_7, x_7.g^1\} \end{aligned}$$

we conclude that $A \vee B$ is equivalent to

$$\{x_2.f^2, x_2.f_2^2 \doteq x_5, x_6 = x_5, x_5.g^1\} \wedge (\{x_1 = x_2, x_4 = x_5\} \vee \{x_3 = x_2, x_7 = x_5\}).$$

Proposition 1 *Let A and B be two solved forms. Then the algorithm computes the sets COM, $\tilde{A}$ and $\tilde{B}$, such that there is no common factor in $\tilde{A}$ and $\tilde{B}$ and such that $A \vee B$ is equivalent to $COM \wedge (\tilde{A} \vee \tilde{B})$.*

Proof. Note that it is quite obvious that $A \vee B$ is equivalent to $COM \wedge (\tilde{A} \vee \tilde{B})$, since A is equivalent to $COM \wedge \tilde{A}$ and B is equivalent to $COM \wedge \tilde{B}$. To prove that there is no common factor left in $\tilde{A}$ and $\tilde{B}$ note that the set $\tilde{A}$, computed by the sets $EQ(A)$, $Proj(A)$ and $Funct(A)$ is such that:

(i) for every $\tilde{X} \in EQ(\tilde{A})$ exists $X \in EQ(A)$ such that $\tilde{X} \subseteq X$;

(ii) if $\tilde{X}.f_i^n = \tilde{Y} \in Proj(\tilde{A})$ then there is $X.f_i^n = Y \in Proj(A)$, such that $\tilde{X} \subseteq X$ and $\tilde{Y} \subseteq Y$;

(iii) if $\tilde{X}.f^n \in Funct(\tilde{A})$ then there is $X.f^n \in Proj(A)$, such that $\tilde{X} \subseteq X$.

Obviously $\tilde{B}$ has the same properties. Now suppose that there is $\tilde{A} \models s = t$ and $\tilde{B} \models s = t$ for some variables or atoms s and t. This means that both have to be in the same equivalence class in $EQ(\tilde{A})$ and in $EQ(\tilde{B})$. But this is impossible by (i) and since simplification step (1) doesn't apply to $EQ(A)$ and $EQ(B)$. In a similar way condition (ii) and the fact that simplification step (2) doesn't apply to $Proj(A)$ and to $Proj(B)$, make it impossible to have $\tilde{A} \models x.f_i^n \doteq s$ and $\tilde{B} \models x.f_i^n \doteq s$. The same reasoning goes for factors of the form $x.f^n$. •

6 Final Remarks

The constraint rewriting system and the low-level implementation presented in this paper have been successfully used in the implementation of a number of grammar formalisms (called Constraint Logic Grammars). Also they were easily modified to deal with feature structures, instead of rational trees. Some improvements in the efficiency of the implementation can be achieved if atomic negated constraints are considered in the solved form (c.f. [DMV92]). We are currently studying the extension of the constraint language to cover constraints over *lists* and *sets*.

References

[DMV91] Lusí Damas, Nelma Moreira, and Giovanni B. Varile. The formal and processing models of CLG. In *Fiifth Conference of the European Chapter of the Association for Computational Linguistics*, pages 173–178, Berlin, 1991.

[DMV92] Luís Damas, Nelma Moreira, and Giovanni B. Varile. The formal and computational theory of constraint logic grammars. In *Proceedings of the Workshop on Constraint Propagation and Linguistic Description*, Lugano, 1992.

[DV89] Luís Damas and Giovanni B. Varile. CLG: A grammar formalism based on constraint resolution. In E.M.Morgado and J.P.Martins, editors, *EPIA 89*, volume 390 of *Lecture Notes in Artificial Intelligence*, pages 175–186. Springer Verlag, 1989.

[DV92] Luís Damas and Giovanni B. Varile. On the satisfiability of complex constraints. In *Proceedings of the 14th International Conference on Computational Linguistics (COLING)*, Nantes, France, 1992.

[ED88] A. Eisele and J. Dörre. Unification of disjunctive feature descriptions. In *26th Annual Meeting of the Association for Computational Linguistics*, Buffalo, New York, 1988.

[Kas87] R. T. Kasper. Unification method for disjunctive feature descriptions. In *25th Annual Meeting of the Association for Computational Linguistics*, Standford, CA, 1987.

[KB82] R. Kaplan and J. Bresnan. Lexical functional grammar: A formal system for grammatical representation. In Joan Bresnan, editor, *The Mental Representation of Grammatical Relations*. MIT Press, 1982.

[Mah88] Michael J. Maher. Complete axiomatizations of the algebras of finite, rational and infinite trees. Technical report, IBM Thomas J. Watson Research Center, P.O. Box 704, Yorktown Heights, NY 10598, U.S.A., 1988.

[MK91] John T. Maxwell and Ronald M. Kaplan. A method for disjunctive constraint satisfaction. In Massaru Tomita, editor, *Current Issues in Parsing Technology*. Kluwer Academic Publishers, 1991.

[PS87] Carl Pollard and Ivan Sag. *Information Based Syntax and Semantics, Volume 1, Fundamentals*, volume 13. Center for the Study of Language and Information Stanford, 1987.

[RK86] W.C. Rounds and R.T. Kasper. A complete logical calculus for record structures representing linguistic information. In *Symposium on Logic in Computer Science*, IEEE Computer Society, 1986.

[Smo89] Gert Smolka. Feature logic with subsorts. Technical report, IBM Wissenschafliches Zentrum, Institut für Wissensbasierte Systeme, 1989. LILOG Report 33.

[ST92] Gert Smolka and Ralf Treinen. Records for logic programming. In Krzysztof Apt, editor, *ICLP92*. MIT, 1992.

[SUP+83] Stuart M. Shieber, Hans Uszkoreit, Fernando C.N. Pereira, J. Robinson, and M. Tyson. The formalism and implementation of PATR-II. In B. J. Grosz and M. E. Stickel, editors, *Research on Interactive Acquisition and Use of Knowledge*. SRI report, 1983.

[Usz86] Hans Uszkoreit. Categorial unification grammar. In *Proceedings of the 11th International Conference on Computational Linguistics (COLING)*, Bonn, 1986.